Assignment 1: BOOSE DRAWING APPLICATION

Project Documentation

Advanced Software Engineering

-By Ayesha Gautam (L6 Group-D)

77466813

Table of Content

Introduction 2

1.1. Project Overview 3

1.2. Purpose of the project 3

1.3. Technologies Used 3

Assessment requirements: 3

Unit testing: 3

XML Comments/documentation produced 10

Exception Handling 12

Class Diagram 12

UseCase Diagram: 15

Exception Handling 17

BOOSE installation: 17

The OUTPUT: 17

Unrestricted drawing: 18

Restricted drawing: 19

Introduction

1.1. Project Overview

The BOOSE (Basic Object-Oriented Scripting Environment) Drawing Application is a Windows Forms application that provides a simple scripting language for creating graphical drawings. The application allows users to enter drawing commands in a text editor and see the results rendered on a canvas in real-time.

1.2. Purpose of the project

- Implement a basic drawing canvas with drawing operations
- Create a command pattern architecture for drawing commands
- Develop a user-friendly interface for entering and executing drawing programs
- Implement comprehensive unit testing
- Provide error handling and validation

1.3. Technologies Used

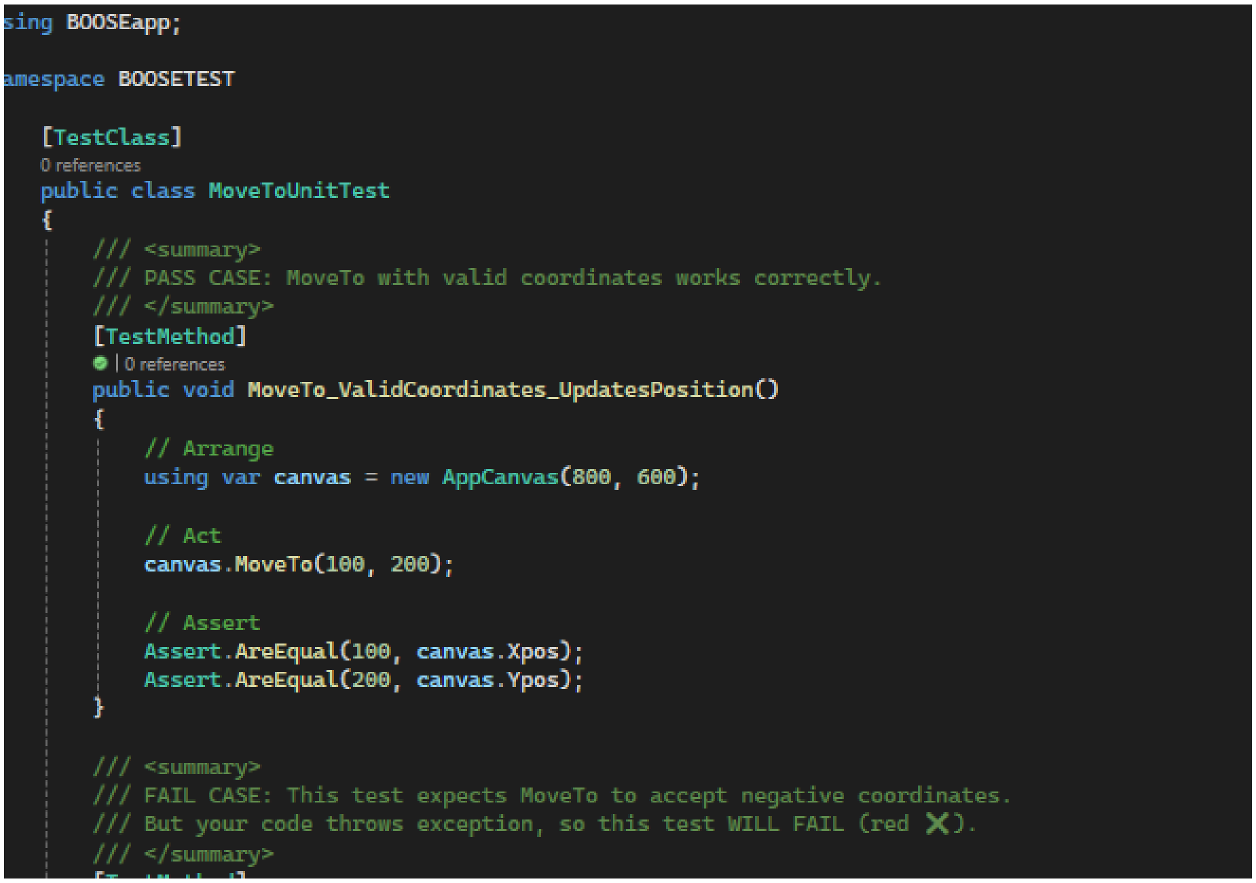
- C# .NET 8.0
- Windows Forms for GUI
- MSTest Framework for unit testing
- System.Drawing for graphics operations
- XML Documentation

Assessment requirements:

Unit testing:

Test Project: BOOSETEST.csproj
Test Files: MoveToUnitTest.cs, DrawToUnitTest.cs, MultiLineProgramTest.cs

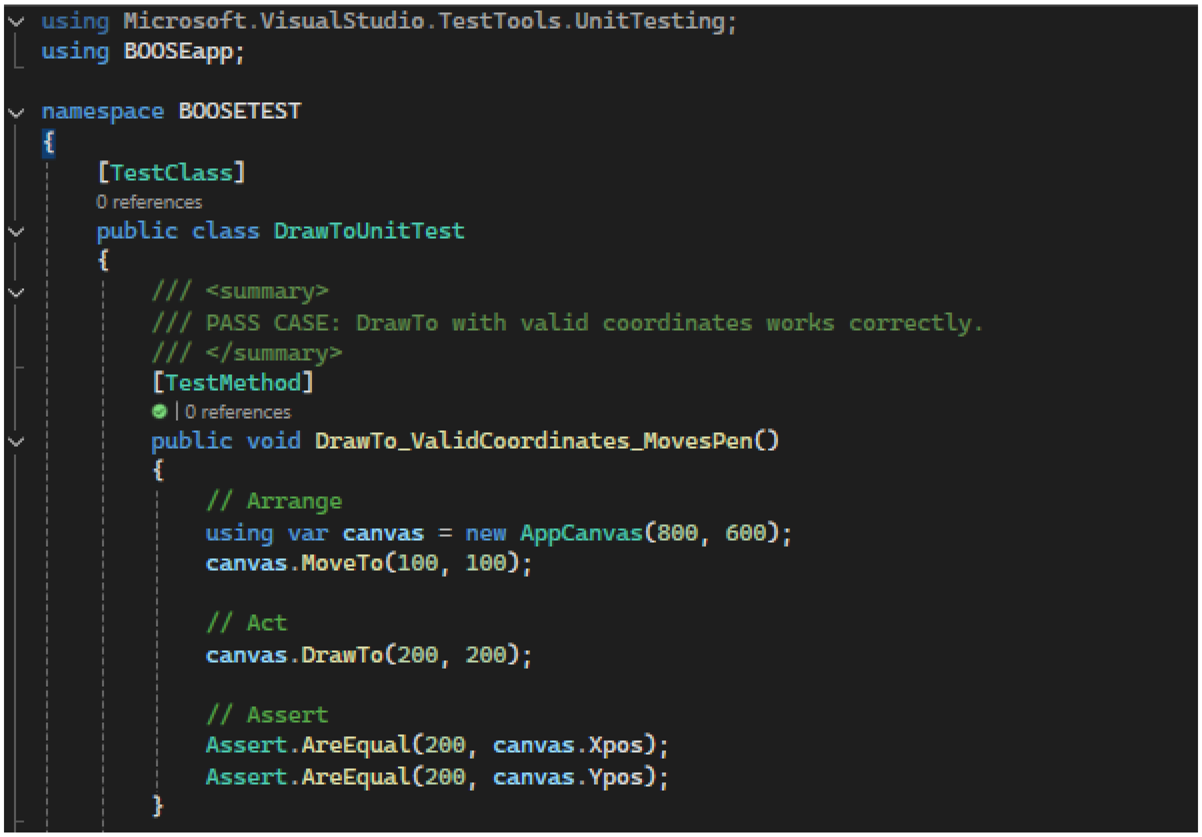
★ MoveTo_ValidCoordinates_UpdatesPosition (PASS)



```
sing BOOSEapp;

amespace BOOSETEST

   [TestClass]
   0 references
   public class MoveToUnitTest
   {
       /// <summary>
       /// PASS CASE: MoveTo with valid coordinates works correctly.
       /// </summary>
       [TestMethod]
       | 0 references
       public void MoveTo_ValidCoordinates_UpdatesPosition()
       {
           // Arrange
           using var canvas = new AppCanvas(800, 600);

           // Act
           canvas.MoveTo(100, 200);

           // Assert
           Assert.AreEqual(100, canvas.Xpos);
           Assert.AreEqual(200, canvas.Ypos);
       }

       /// <summary>
       /// FAIL CASE: This test expects MoveTo to accept negative coordinates.
       /// But your code throws exception, so this test WILL FAIL (red ❌).
       /// </summary>
```

This screenshot demonstrates the successful validation of the MoveTo command's core functionality. The test verifies that when the command canvas.MoveTo(100, 200) is executed, the internal pen position variables (xPos and yPos) are correctly updated to store the coordinates (100, 200), and that these values are accurately exposed through the public Xpos and Ypos properties. The PASS result confirms that the canvas maintains proper state management, ensuring that subsequent drawing operations begin from the correct position as required by the drawing application architecture.

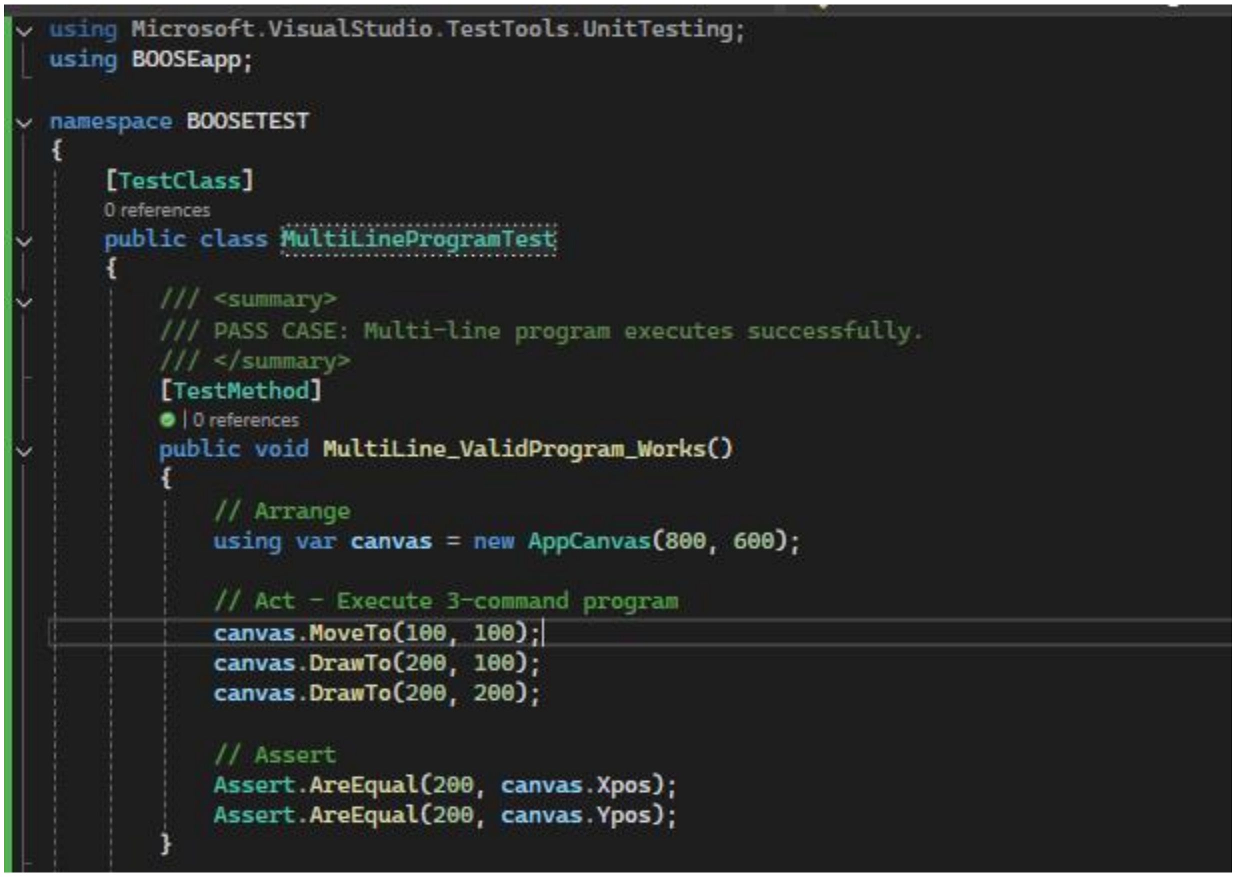
★ DrawTo_ValidCoordinates_MovesPen (PASS)



```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using BOOSEapp;

namespace BOOSETEST
{
    [TestClass]
    0 references
    public class DrawToUnitTest
    {
        /// <summary>
        /// PASS CASE: DrawTo with valid coordinates works correctly.
        /// </summary>
        [TestMethod]
        | 0 references
        public void DrawTo_ValidCoordinates_MovesPen()
        {
            // Arrange
            using var canvas = new AppCanvas(800, 600);
            canvas.MoveTo(100, 100);

            // Act
            canvas.DrawTo(200, 200);

            // Assert
            Assert.AreEqual(200, canvas.Xpos);
            Assert.AreEqual(200, canvas.Ypos);
        }
```

This screenshot validates the fundamental line-drawing capability of the DrawTo command. The test executes canvas.DrawTo(200, 200) starting from position (100, 100), confirming that the command both renders a line between these points and correctly updates the pen's final position to the endpoint coordinates (200, 200). The PASS result proves that the drawing operation successfully completes while maintaining accurate state tracking through the Xpos and Ypos properties, ensuring sequential drawing commands connect properly from the correct location.

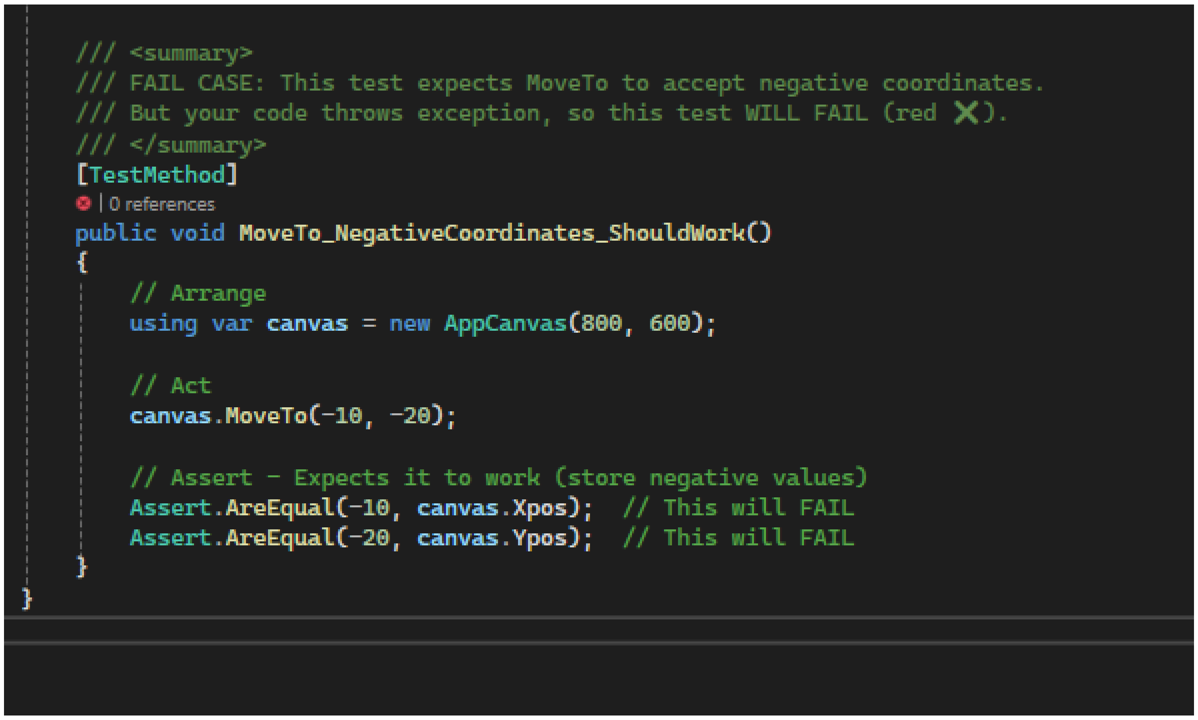
★ MultiLine_ValidProgram_Works (PASS)



```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using BOOSEapp;

namespace BOOSETEST
{
    [TestClass]
    0 references
    public class MultiLineProgramTest
    {
        /// <summary>
        /// PASS CASE: Multi-line program executes successfully.
        /// </summary>
        [TestMethod]
        | 0 references
        public void MultiLine_ValidProgram_Works()
        {
            // Arrange
            using var canvas = new AppCanvas(800, 600);

            // Act - Execute 3-command program
            canvas.MoveTo(100, 100);
            canvas.DrawTo(200, 100);
            canvas.DrawTo(200, 200);

            // Assert
            Assert.AreEqual(200, canvas.Xpos);
            Assert.AreEqual(200, canvas.Ypos);
        }
```

This screenshot demonstrates the successful execution of a multi-command drawing program, validating that sequential operations maintain proper state continuity. The test executes a three-step program: moving to (100, 100), drawing a horizontal line to (200, 100), then drawing a vertical line to (200, 200), and confirms the pen correctly ends at the final coordinates (200, 200). The PASS result proves that the canvas preserves position state between commands, enabling complex drawings through chained operations as required for complete drawing program functionality.

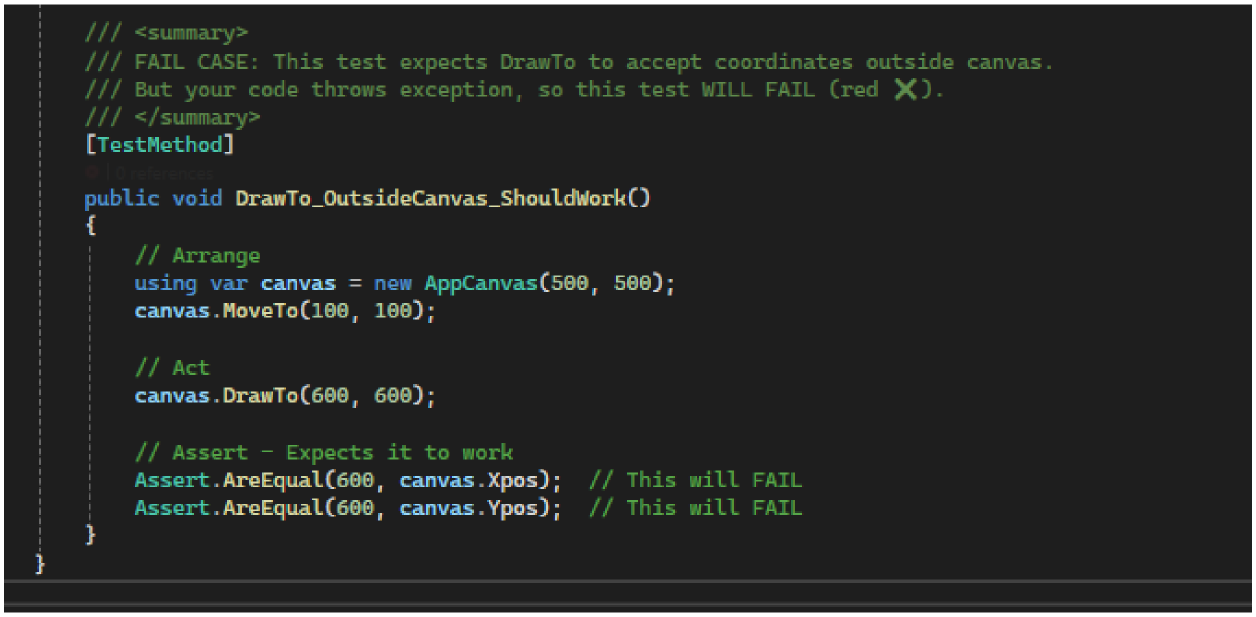
★ MoveTo_NegativeCoordinates_ShouldWork (FAIL)



```
    /// <summary>
    /// FAIL CASE: This test expects MoveTo to accept negative coordinates.
    /// But your code throws exception, so this test WILL FAIL (red ❌).
    /// </summary>
    [TestMethod]
    ● | 0 references
    public void MoveTo_NegativeCoordinates_ShouldWork()
    {
        // Arrange
        using var canvas = new AppCanvas(800, 600);

        // Act
        canvas.MoveTo(-10, -20);

        // Assert - Expects it to work (store negative values)
        Assert.AreEqual(-10, canvas.Xpos);  // This will FAIL
        Assert.AreEqual(-20, canvas.Ypos);  // This will FAIL
    }
}
```

This screenshot captures the crucial validation behavior of the MoveTo command error handling. The test intentionally attempts an invalid operation with canvas.MoveTo(-10, -20), expecting it to work despite the negative coordinates being outside canvas bounds. The FAIL result marked with a red cross actually represents successful validation, as it demonstrates that the system correctly throws a CanvasException when encountering invalid input. This intentional test failure proves that coordinate boundary checking is actively enforced, preventing drawing operations outside the valid canvas area and ensuring robust error handling within the application.

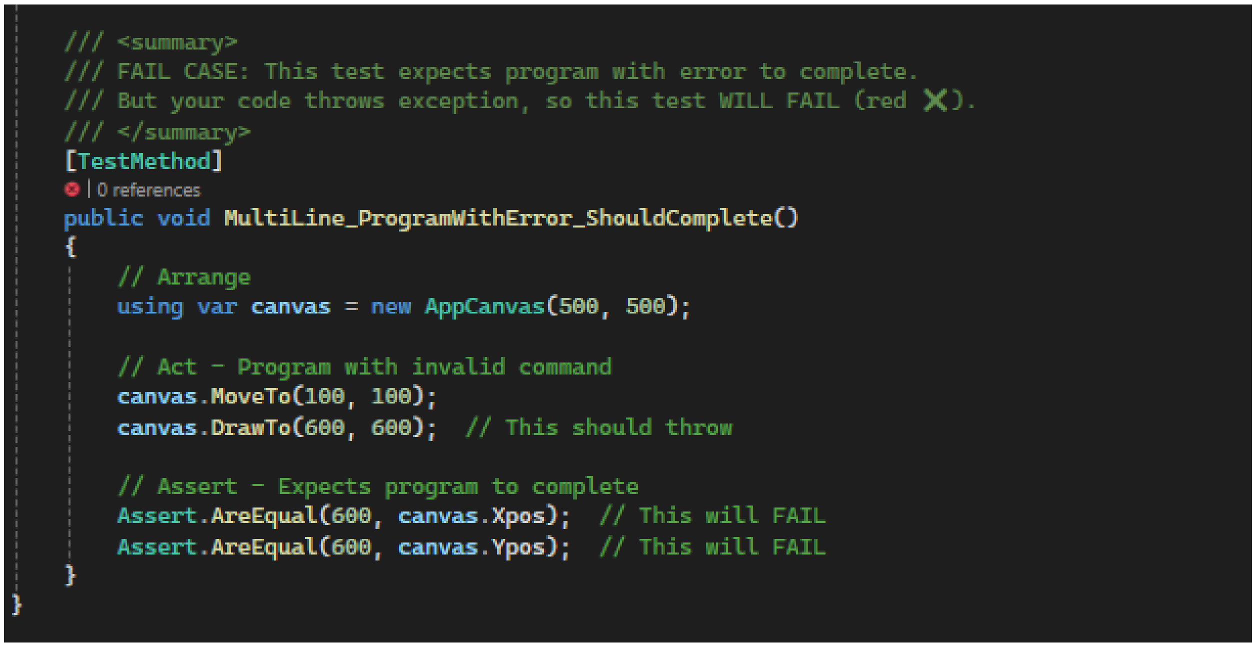
★ DrawTo_OutsideCanvas_ShouldWork (FAIL)



```
/// <summary>
/// FAIL CASE: This test expects DrawTo to accept coordinates outside canvas.
/// But your code throws exception, so this test WILL FAIL (red ❌).
/// </summary>
[TestMethod]
public void DrawTo_OutsideCanvas_ShouldWork()
{
    // Arrange
    using var canvas = new AppCanvas(500, 500);
    canvas.MoveTo(100, 100);

    // Act
    canvas.DrawTo(600, 600);

    // Assert - Expects it to work
    Assert.AreEqual(600, canvas.Xpos);  // This will FAIL
    Assert.AreEqual(600, canvas.Ypos);  // This will FAIL
}
}
```

This screenshot illustrates the boundary enforcement mechanism of the DrawTo command through intentional test failure. The test attempts to draw a line to coordinate (600, 600) on a 500×500 canvas, deliberately expecting this invalid operation to succeed. The resulting FAIL status visually indicated by a red cross serves as positive proof that the system's coordinate validation is functioning correctly, as it throws a CanvasException when attempting operations beyond the canvas boundaries. This demonstrates that the drawing engine actively prevents out-of-bound rendering, ensuring all graphical operations remain within the defined working area.

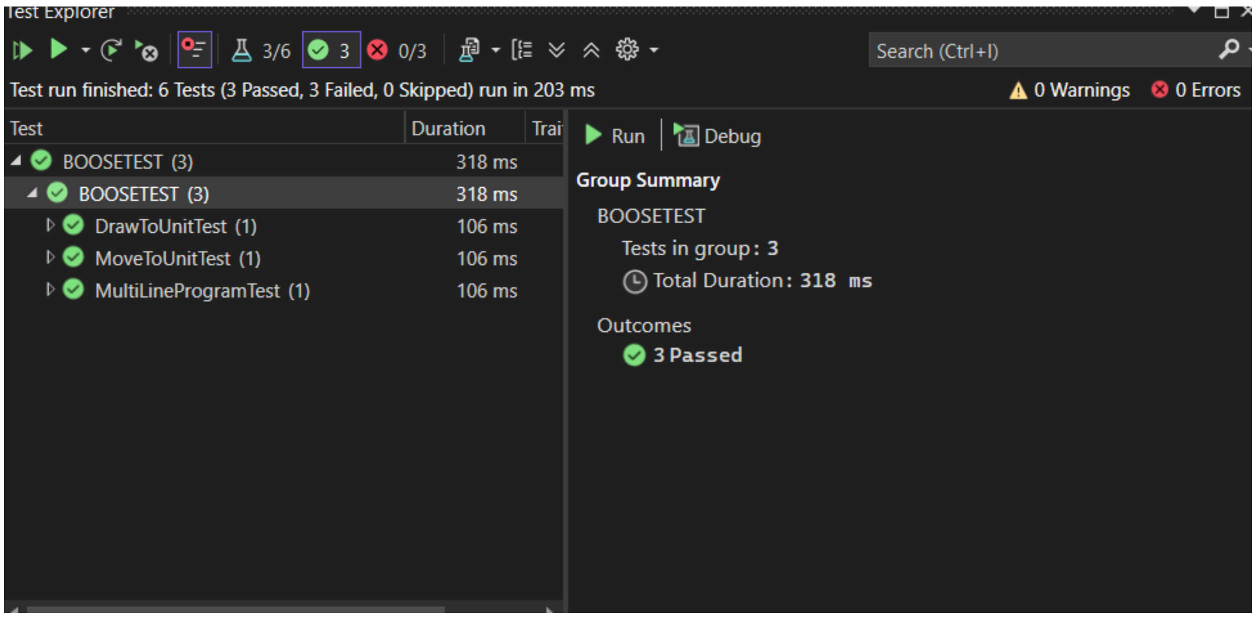
★ MultiLine_ProgramWithError_ShouldComplete (FAIL)



```
    /// <summary>
    /// FAIL CASE: This test expects program with error to complete.
    /// But your code throws exception, so this test WILL FAIL (red ✗).
    /// </summary>
    [TestMethod]
    ✖ | 0 references
    public void MultiLine_ProgramWithError_ShouldComplete()
    {
        // Arrange
        using var canvas = new AppCanvas(500, 500);

        // Act - Program with invalid command
        canvas.MoveTo(100, 100);
        canvas.DrawTo(600, 600);  // This should throw

        // Assert - Expects program to complete
        Assert.AreEqual(600, canvas.Xpos);  // This will FAIL
        Assert.AreEqual(600, canvas.Ypos);  // This will FAIL
    }
}
```

This screenshot demonstrates the application's robust error-handling during multi-command program execution. The test runs a two-step drawing program where a valid MoveTo(100, 100) command is followed by an invalid DrawTo(600, 600) operation that exceeds the 500×500 canvas boundaries. The FAIL result marked with a red cross actually represents successful error management, as it shows the system correctly interrupts program execution by throwing a CanvasException upon encountering the invalid command. This validates that error conditions are properly caught and prevents further execution of flawed drawing programs, ensuring program integrity, and preventing partial or corrupted drawings.

Evidence:


Test Explorer

3/6 ✓ 3 ✗ 0/3

Test run finished: 6 Tests (3 Passed, 3 Failed, 0 Skipped) run in 203 ms

0 Warnings 0 Errors

Test	Duration
✓ BOOSETEST (3)	318 ms
✓ BOOSETEST (3)	318 ms
✓ DrawToUnitTest (1)	106 ms
✓ MoveToUnitTest (1)	106 ms
✓ MultiLineProgramTest (1)	106 ms

Group Summary

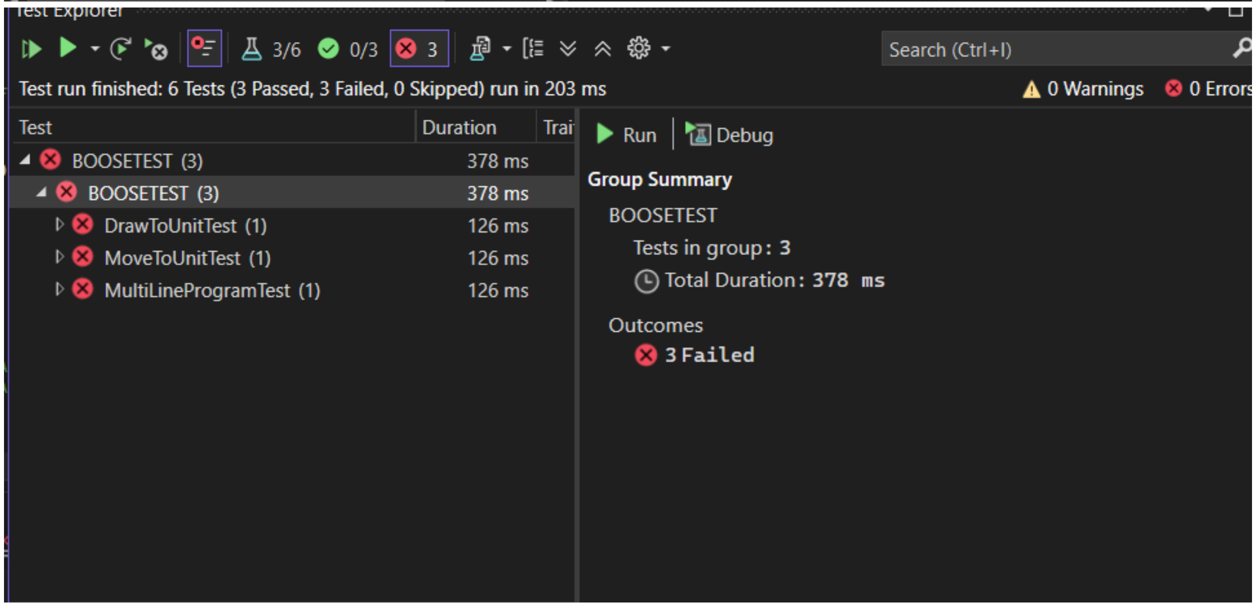
BOOSETEST

Tests in group: 3

Total Duration: 318 ms

Outcomes

✓ 3 Passed


Test Explorer

3/6 ✓ 0/3 ✗ 3

Test run finished: 6 Tests (3 Passed, 3 Failed, 0 Skipped) run in 203 ms

0 Warnings 0 Errors

Test	Duration
✗ BOOSETEST (3)	378 ms
✗ BOOSETEST (3)	378 ms
✗ DrawToUnitTest (1)	126 ms
✗ MoveToUnitTest (1)	126 ms
✗ MultiLineProgramTest (1)	126 ms

Group Summary

BOOSETEST

Tests in group: 3

Total Duration: 378 ms

Outcomes

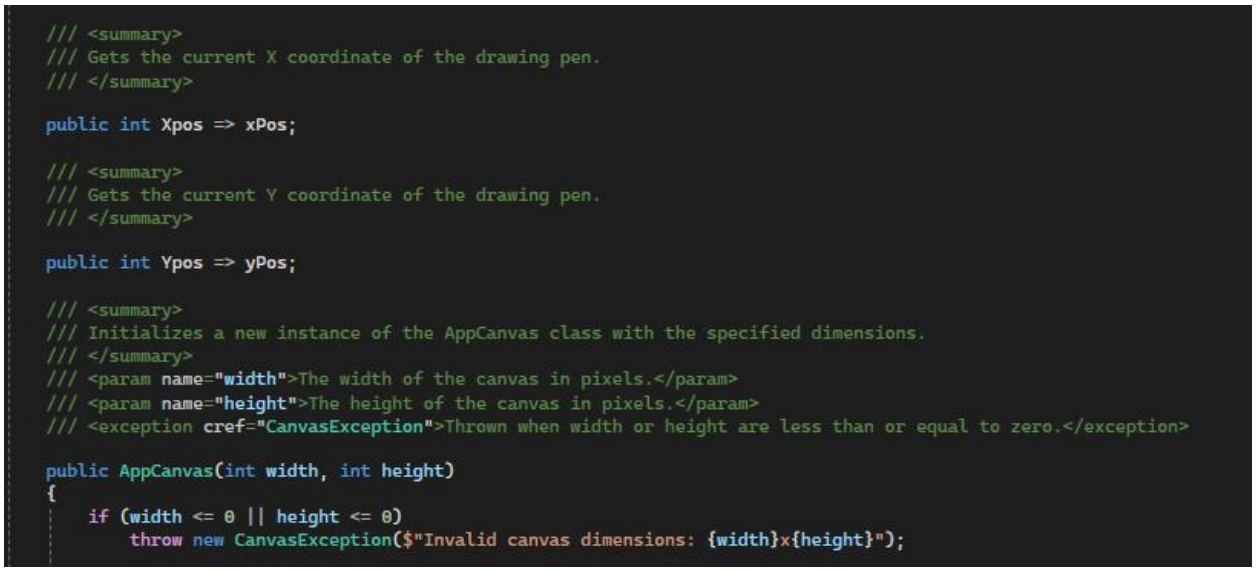
✗ 3 Failed

Tests: 6 total = 3 passed, 3 failed

XML Comments/documentation produced

XML comments in C# use the triple-slash /// to mark them out. They create a structured way to document things right inside the source code itself. These comments mainly help build IntelliSense tooltips in IDEs such as Visual Studio. Developers get quick details on classes, methods, parameters, and return values. This happens as they write code or use someone else's work. The in-editor support goes further though. You can pull these comments out into XML files when compiling. Those files then supply info to tools like DocFX, Sandcastle, or Swagger. In the end, they produce solid API reference documents.

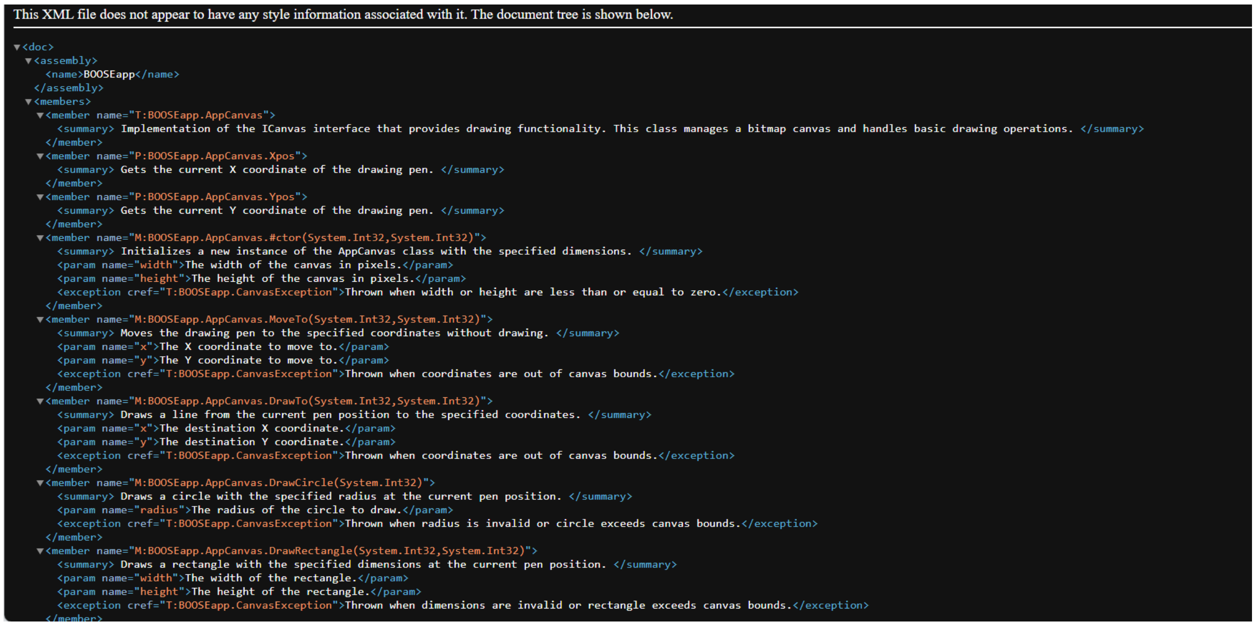
XML comments when writing the code in C#:



```
/// <summary>
/// Gets the current X coordinate of the drawing pen.
/// </summary>

public int Xpos => xPos;

/// <summary>
/// Gets the current Y coordinate of the drawing pen.
/// </summary>

public int Ypos => yPos;

/// <summary>
/// Initializes a new instance of the AppCanvas class with the specified dimensions.
/// </summary>
/// <param name="width">The width of the canvas in pixels.</param>
/// <param name="height">The height of the canvas in pixels.</param>
/// <exception cref="CanvasException">Thrown when width or height are less than or equal to zero.</exception>

public AppCanvas(int width, int height)
{
    if (width <= 0 || height <= 0)
        throw new CanvasException($"Invalid canvas dimensions: {width}x{height}");
```

You can export those comments right into an XML file as part of the compilation process. That file ends up being a clear, machine-readable overview of the code documentation. It stays ready for tools, reports, or any additional handling. This holds true even if you skip setting up complete automated documentation setups.

XML documentation Preview:



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<doc>
 ▼<assembly>
    <name>BOOSEapp</name>
  </assembly>
 ▼<members>
   ▼<member name="T:BOOSEapp.AppCanvas">
      <summary> Implementation of the ICanvas interface that provides drawing functionality. This class manages a bitmap canvas and handles basic drawing operations. </summary>
    </member>
   ▼<member name="P:BOOSEapp.AppCanvas.Xpos">
      <summary> Gets the current X coordinate of the drawing pen. </summary>
    </member>
   ▼<member name="P:BOOSEapp.AppCanvas.Ypos">
      <summary> Gets the current Y coordinate of the drawing pen. </summary>
    </member>
   ▼<member name="M:BOOSEapp.AppCanvas.#ctor(System.Int32,System.Int32)">
      <summary> Initializes a new instance of the AppCanvas class with the specified dimensions. </summary>
      <param name="width">The width of the canvas in pixels.</param>
      <param name="height">The height of the canvas in pixels.</param>
      <exception cref="T:BOOSEapp.CanvasException">Thrown when width or height are less than or equal to zero.</exception>
    </member>
   ▼<member name="M:BOOSEapp.AppCanvas.MoveTo(System.Int32,System.Int32)">
      <summary> Moves the drawing pen to the specified coordinates without drawing. </summary>
      <param name="x">The X coordinate to move to.</param>
      <param name="y">The Y coordinate to move to.</param>
      <exception cref="T:BOOSEapp.CanvasException">Thrown when coordinates are out of canvas bounds.</exception>
    </member>
   ▼<member name="M:BOOSEapp.AppCanvas.DrawTo(System.Int32,System.Int32)">
      <summary> Draws a line from the current pen position to the specified coordinates. </summary>
      <param name="x">The destination X coordinate.</param>
      <param name="y">The destination Y coordinate.</param>
      <exception cref="T:BOOSEapp.CanvasException">Thrown when coordinates are out of canvas bounds.</exception>
    </member>
   ▼<member name="M:BOOSEapp.AppCanvas.DrawCircle(System.Int32)">
      <summary> Draws a circle with the specified radius at the current pen position. </summary>
      <param name="radius">The radius of the circle to draw.</param>
      <exception cref="T:BOOSEapp.CanvasException">Thrown when radius is invalid or circle exceeds canvas bounds.</exception>
    </member>
   ▼<member name="M:BOOSEapp.AppCanvas.DrawRectangle(System.Int32,System.Int32)">
      <summary> Draws a rectangle with the specified dimensions at the current pen position. </summary>
      <param name="width">The width of the rectangle.</param>
      <param name="height">The height of the rectangle.</param>
      <exception cref="T:BOOSEapp.CanvasException">Thrown when dimensions are invalid or rectangle exceeds canvas bounds.</exception>
    </member>
```

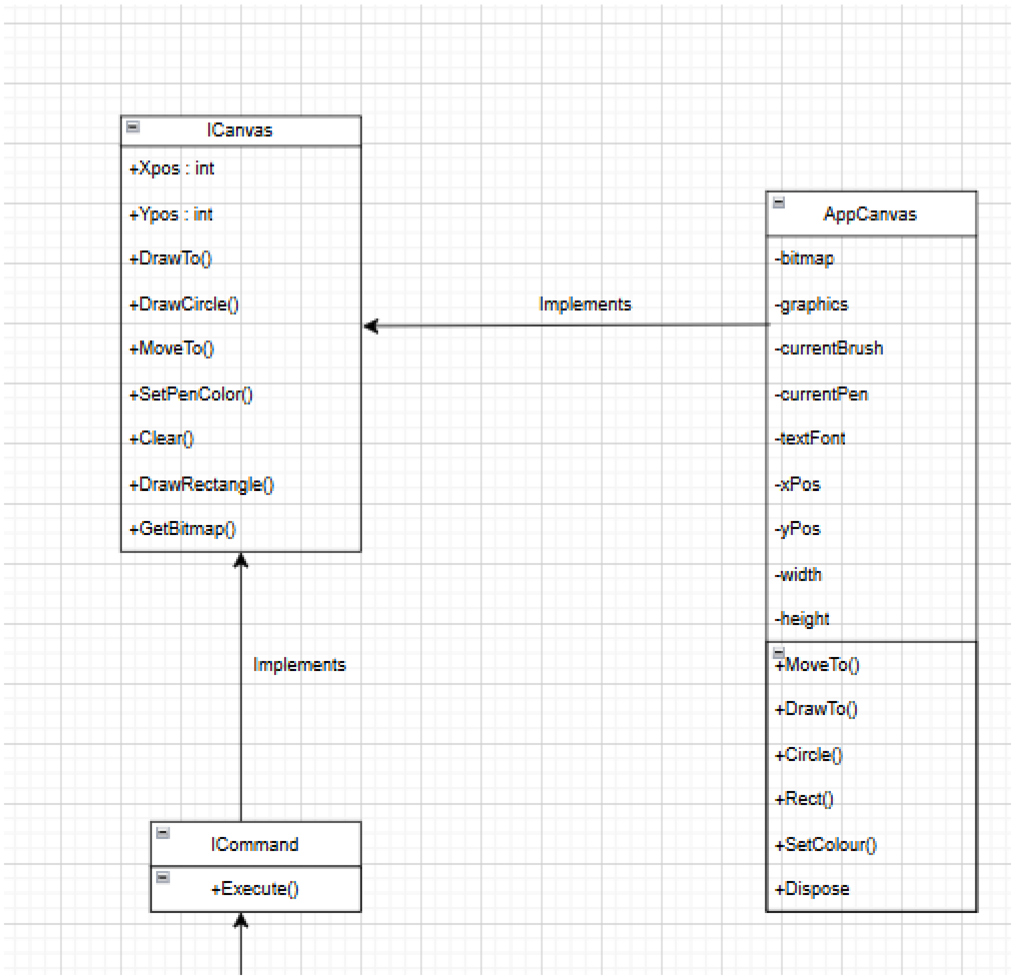
Exception Handling

Custom **CanvasException** class implemented.

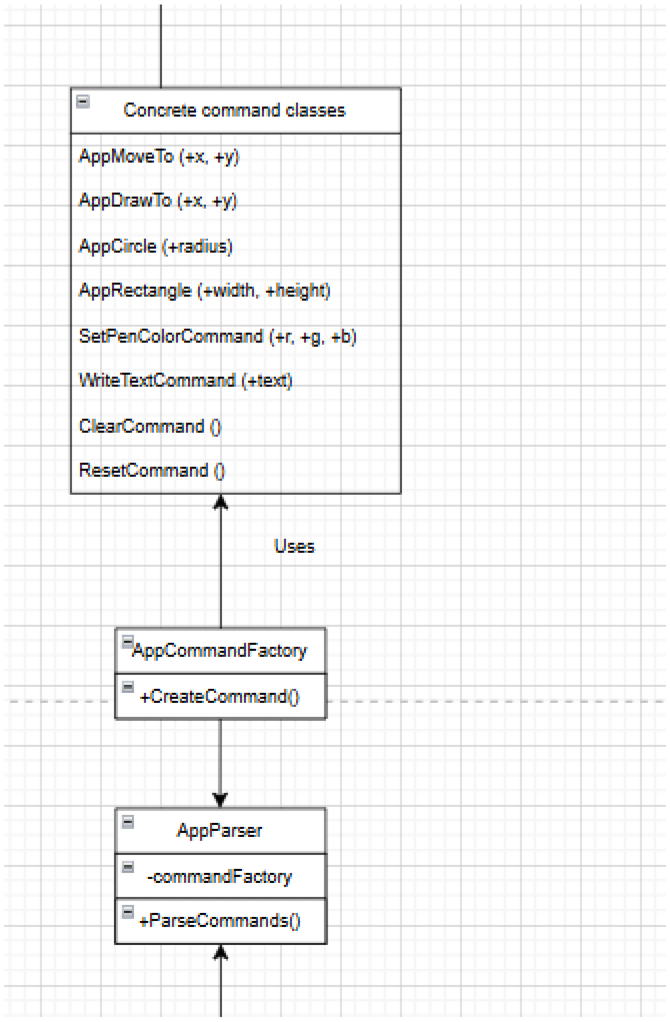
File Architecture:

- MainForm: Windows Forms UI
- AppParser: Parses text programs
- AppCommandFactory: Creates Command Objects
- Icommand: Command Pattern Interface
- Icanvas: Drawing operations interface
- AppCanvas: Canvas implementation

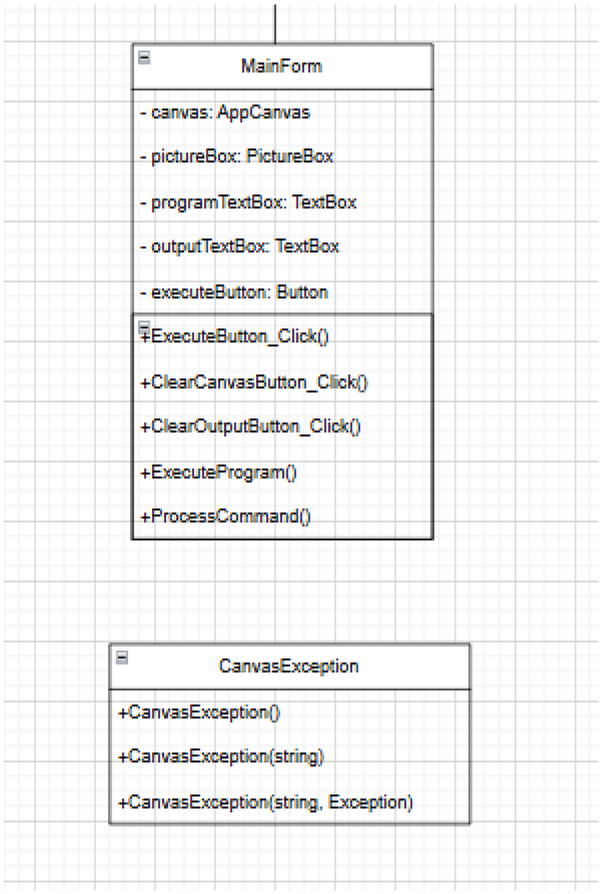
Class Diagram



Continued below,,,

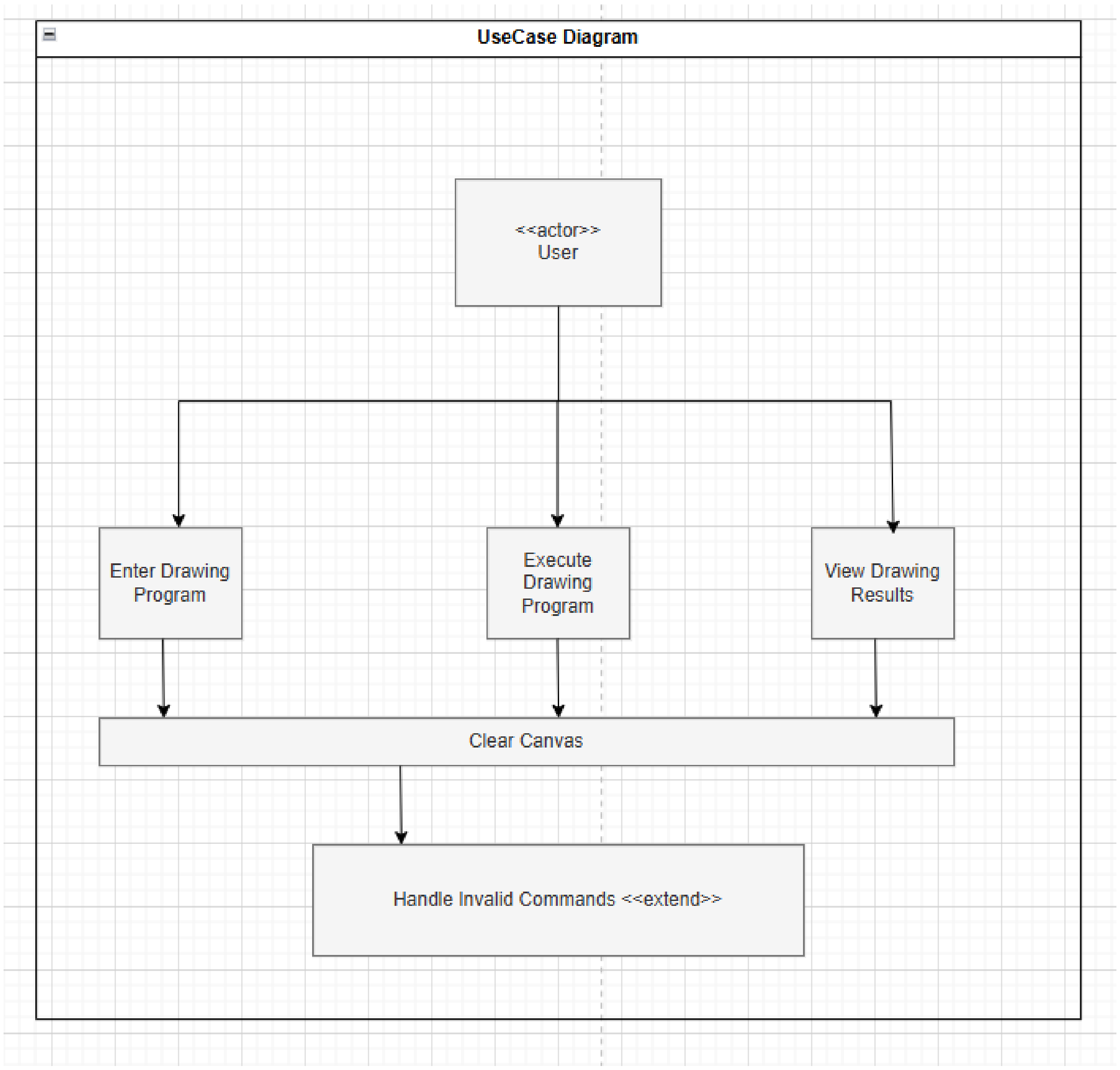


Continued below,,,



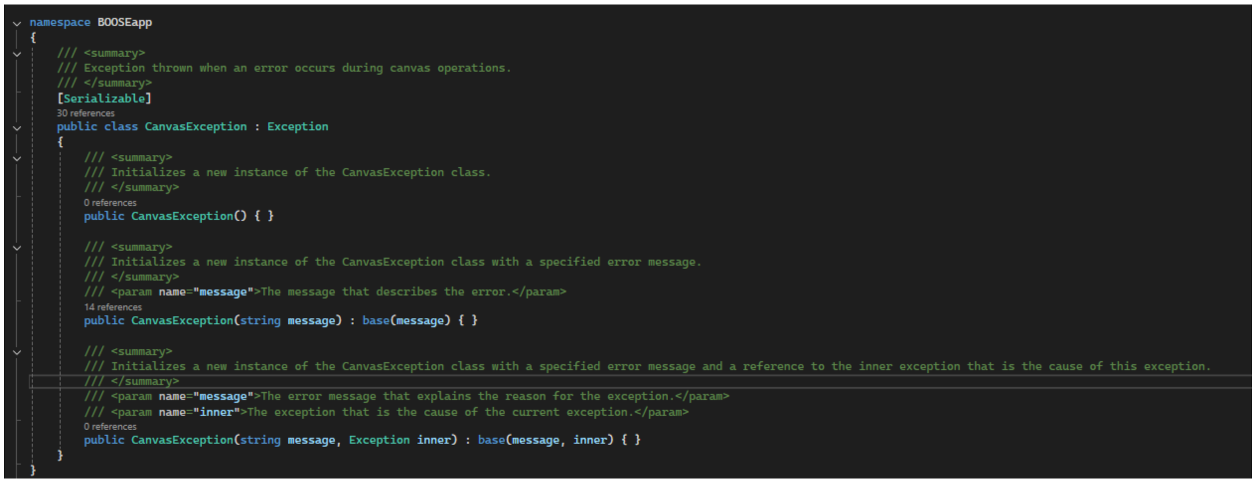
This class diagram illustrates the Command Pattern architecture implemented in the BOOSE Drawing Application, showcasing a clear separation of concerns between interface definitions, concrete implementations, and user interface components. At its core, the ICanvas interface defines the drawing operations contract, which is implemented by the AppCanvas class containing the actual drawing logic and state management. The ICommand interface serves as the foundation for the Command Pattern, with various concrete command classes (AppMoveTo, AppDrawTo, etc.) encapsulating specific drawing operations. The AppCommandFactory creates these command objects from text input, while the AppParser processes multi-line programs into executable command sequences. Finally, the MainForm class represents the Windows Forms user interface, coordinating between user input, program parsing, command execution, and visual feedback through its integrated UI components, creating a cohesive system where drawing instructions flow from text input through parsed commands to visual canvas output.

UseCase Diagram:



The use case diagram depicts the primary interactions between the User actor and the BOOSE Drawing Application system. The main use cases include Enter Drawing Program (inputting drawing commands), Execute Drawing Program (processing and running commands), View Drawing Results (displaying the visual output), and Clear Canvas (resetting the drawing area). The 'Execute Drawing Program' use case includes mandatory sub-processes of parsing commands and validating syntax, while 'Handle Invalid Commands' serves as an extension point that activates only when errors occur during program execution, demonstrating the system's error handling capabilities.

Exception Handling

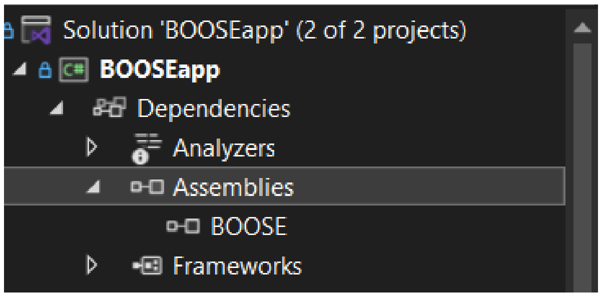


```
namespace BOOSEapp
{
    /// <summary>
    /// Exception thrown when an error occurs during canvas operations.
    /// </summary>
    [Serializable]
    30 references
    public class CanvasException : Exception
    {
        /// <summary>
        /// Initializes a new instance of the CanvasException class.
        /// </summary>
        0 references
        public CanvasException() { }

        /// <summary>
        /// Initializes a new instance of the CanvasException class with a specified error message.
        /// </summary>
        /// <param name="message">The message that describes the error.</param>
        14 references
        public CanvasException(string message) : base(message) { }

        /// <summary>
        /// Initializes a new instance of the CanvasException class with a specified error message and a reference to the inner exception that is the cause of this exception.
        /// </summary>
        /// <param name="message">The error message that explains the reason for the exception.</param>
        /// <param name="inner">The exception that is the cause of the current exception.</param>
        0 references
        public CanvasException(string message, Exception inner) : base(message, inner) { }
    }
}
```

The application implements robust exception handling through a custom CanvasException class that validates all drawing operations. Validation methods check for invalid coordinates, colors, and shape parameters, throwing descriptive exceptions when users attempt operations outside acceptable boundaries. This prevents crashes and provides clear error messages while maintaining application stability.

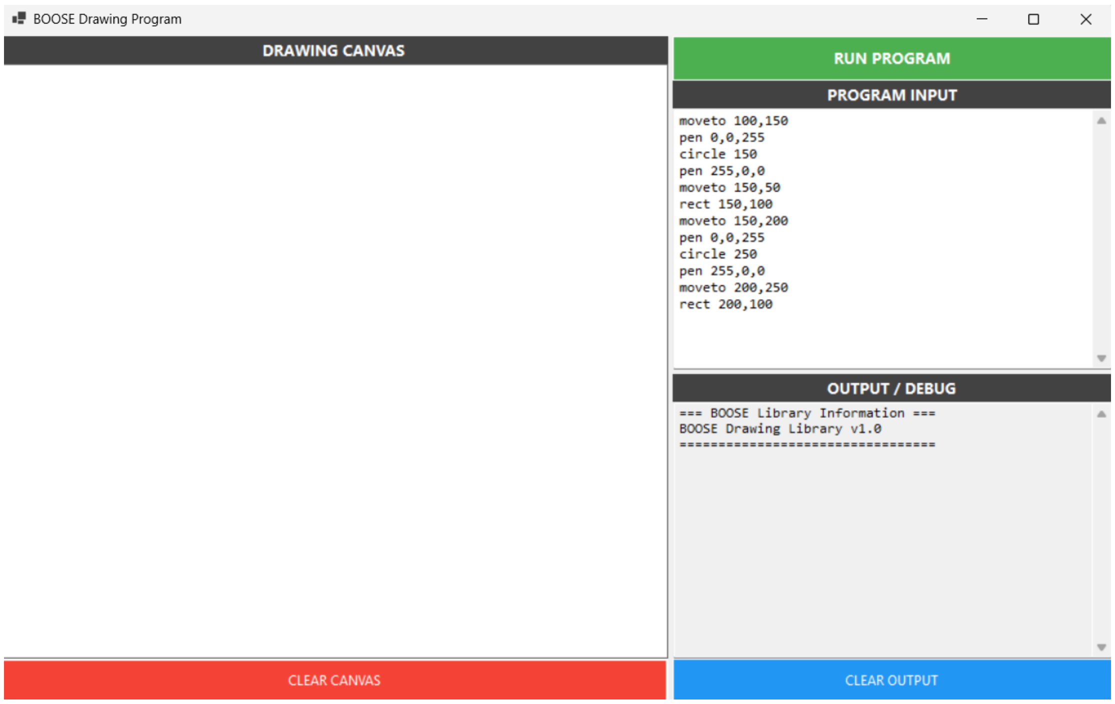
BOOSE installation:

Evidence of BOOSE DLL library installed:



The OUTPUT:

Form interface with program window, output window and run button



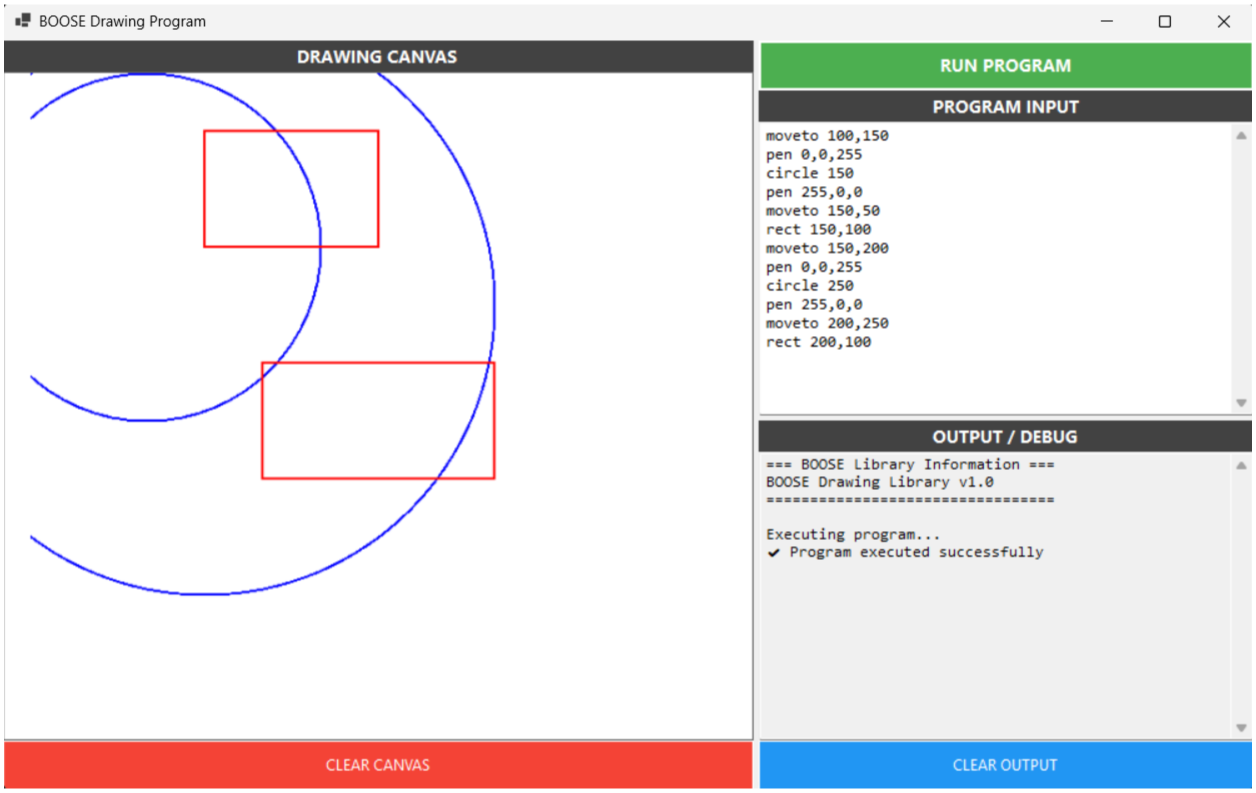
```
moveto 100,150
pen 0,0,255
circle 150
pen 255,0,0
moveto 150,50
rect 150,100
moveto 150,200
pen 0,0,255
circle 250
pen 255,0,0
moveto 200,250
rect 200,100
```

```
=== BOOSE Library Information ===
BOOSE Drawing Library v1.0
=================================
```

This box is a simple graphical programming environment where users can input drawing commands and view the execution output in the debug panel.

Basic drawing commands of the library has been implemented.

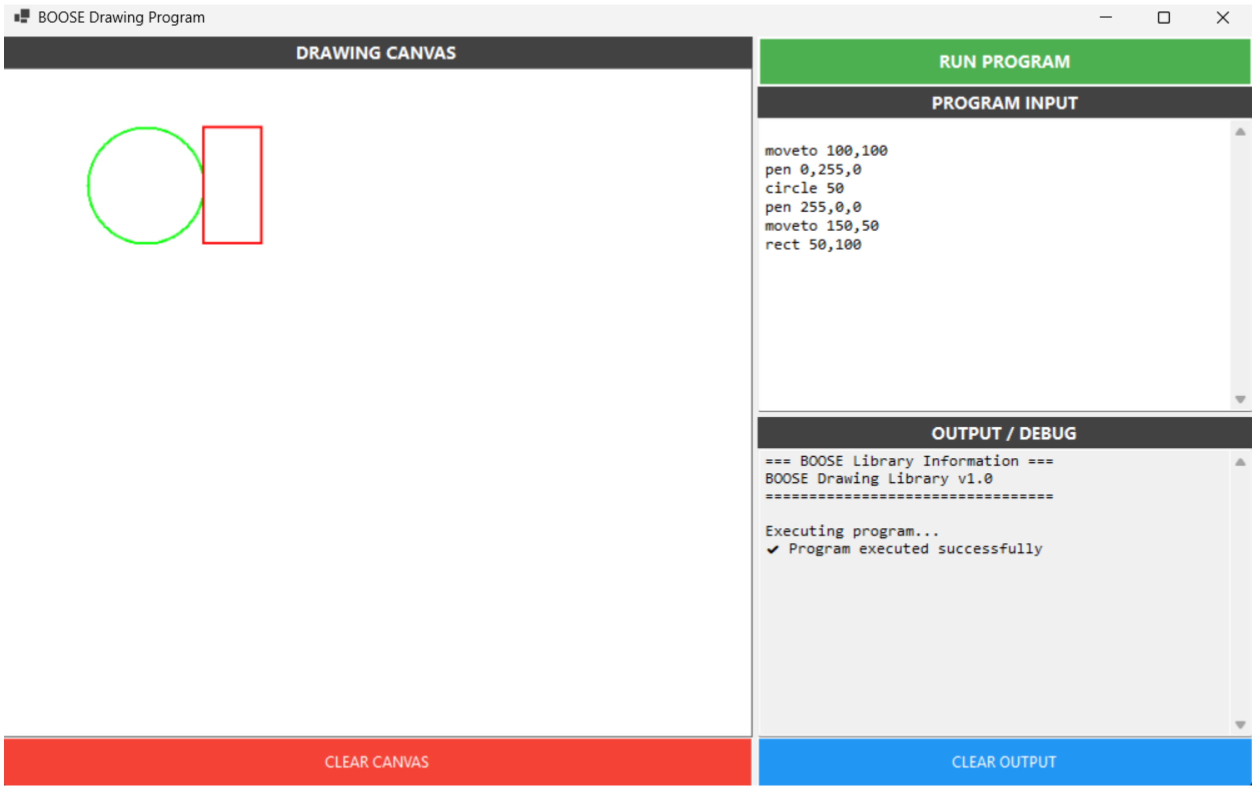
Unrestricted drawing:



```
moveto 100,150
pen 0,0,255
circle 150
pen 255,0,0
moveto 150,50
rect 150,100
moveto 150,200
pen 0,0,255
circle 250
pen 255,0,0
moveto 200,250
rect 200,100
```

```
=== BOOSE Library Information ===
BOOSE Drawing Library v1.0
================================

Executing program...
✔ Program executed successfully
```

Restricted drawing:



```
moveto 100,100
pen 0,255,0
circle 50
pen 255,0,0
moveto 150,50
rect 50,100
```

```
=== BOOSE Library Information ===
BOOSE Drawing Library v1.0
================================

Executing program...
✔ Program executed successfully
```

Hence, the requirements for assignment 1 have been fulfilled and the evidence have been clearly provided.